AD-A241 501

NTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 25 Mar 1991 to 01 Jun 1993

4. TITLE AND SUBTITLE
TeleSoft, TeleGen2 Ada Cross Development System, Version 3.1, for SPARC to 68k, SUN-4/60 (SPARC)(Host) to Motorola MVME-147 (MC68030(Target), 910325i1.11140

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, Federal Republic of Germany

DTIC ELECTE

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IABG-AVF, Industrieanlagen-Betriebsgesellschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION REPORT NUMBER
VSR-VSR 093

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
TeleSoft, TeleGen2 Ada Cross Development System, Version 3.1, for SPARC to 68k, Ottobrunn, Germany, SUN-4/60 (SPARC)(Host) to Motorola MVME-147 (MC68030(Target), ACVC 1.11.

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val Summary Report, Ada Compiler Val Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 91-03-25.

Compiler Name and Version:	TeleGen2™ Ada Cross Development System, Version 3.1, for SPARC to 68K
Host Computer System:	SUN-4/60 (SPARC) under SunOS 4.1
Target Computer System:	Motorola MVME-147 (MC68030) with TeleAda-ZIBC runtime, Version 1.0

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate #910325I1.11140 is awarded to TeleSoft. This certificate expires on 01 March 1993.

This report has been reviewed and is approved.

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

AVF Control Number: IABG-VSR C93
25 March 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910325I1.11140
TeleSoft
TeleGen2™ Ada Cross Development System
Version 3.1, for SPARC to 68K
SUN-4/60 (SPARC) =>
Motorola MVME-147 (MC68030)

== based on TEMPLATE Version 91-01-10 ==

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Declaration of Conformance

Customer: Telesoft
Ada Validation Facility: IABG
ACVC Version: 1.11

Ada Implementation

Ada Compiler Name: TeleGen2™ Ada Cross Development System,
Version 3.1 for SPARC to 68K
Host Computer System: Sun-4/60 under Sun OS 4.1
Target Computer System: Motorola MVME147 (MC68030) board
with TeleAda-EXEC runtime

Customer's Declaration:

I the undersigned representing Telesoft declare that Telesoft has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A, ISO 8652-1987, in the implementation listed in this declaration.

Date: March 22, 1991

Stefan Björnson

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1 1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. §552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institu e for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs: performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 91-03-14.

E28005C	B28006C	C34006D	C35508I	C35508J	C35508M
C35508N	C35702A	C35702B	B41308B	C43004A	C45114A
C45346A	C45612A	C45612B	C45612C	C45651A	C46022A
B49008A	A74006A	C74308A	B83022B	B83022H	B83025B
B83025D	B83026B	C83026A	C83041A	B85001L	C86001F
C94021A	C97116A	C98003B	BA2011A	CB7001A	CB7001B
CB7004A	CC1223A	BC1226A	CC1226B	BC3009B	BD1B02B
BD1B06A	AD1B08A	BD2A02A	CD2A21E	CD2A23E	CD2A32A
CD2A41A	CD2A41E	CD2A87A	CD2B15C	BD3006A	BD4008A
CD4022A	CD4022D	CD4024B	CD4024C	CD4024D	CD4031A
CD4051D	CD5111A	CD7004C	ED7005D	CD7005E	AD7006A
CD7006E	AD7201A	AD7201E	CD7204B	AD7206A	BD8002A
BD8004C	CD9005A	CD9005B	CDA201E	CE2107I	CE2117A
CE2117B	CE2119B	CE2205B	CE2405A	CE3111C	CE3116A
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 21 tests check for the predefined type SHORT_INTEGER:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C45504E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater.

C46013B, C46031B, C46033B, and C46034B contain 'SMALL representation clauses which are not powers of an integer.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types; for this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y checks for a predefined fixed-point type other than DURATION.

CA2009C, CA2009F, BC3204C, and BC3205D check whether a generic unit can be instantiated BEFORE its generic body (and any of its subunits) is compiled. This implementation creates a dependence on generic units as allowed by AI-00408 and AI-00530 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3)

LA3004B, EA3004D, and CA3004F check for pragma INLINE for functions.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

AE2101C and EE2201D..E (2 tests) use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2107B..E (4 tests), CE2107L, and CE2110B attempt to associate multiple internal files with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

CE2107G..H (2 tests), CE2110D, and CE2111H attempt to associate multiple internal files with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

CE2111D checks the resetting of an external file from IN_FILE to OUT_FILE.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A attempt to associate multiple internal files with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 12 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B71001Q	BA3006A	BA3006B	BA3007B	BA3008A
BA3008B	BA3013A			

CA2009C, CA2009F, BC3204C, and BC3205D were graded inapplicable by Evaluation Modification as directed by the AVO. Because the implementation makes the units with instantiations obsolete (see section 2.2), the Class C tests were rejected at link time and the Class B tests were compiled without error.

Test CD21A21C prints the message

```
* NO_NAME CHECK OF REPRESENTATION FOR PRIVATE_ENUM FAILED
```

before printing any regular test output. The AVO ruled that this test may be graded PASSED by Evaluation Modification, because the test is erroneous.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for both technical and sales information about this Ada implementation system, see:

TeleSoft Europe
Bryggargatan 6
P.O. Box 1001
S-14901 Nynäshamn
Sweden

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3769	
b) Total Number of Withdrawn Tests	93	
c) Processed Inapplicable Tests	107	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	308	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 308 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by Ethernet, and run. The results were captured on the host computer system.

Test output, compiler and linker listings, and job logs were captured on a magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test are given on the next page, which was supplied by the customer.

Basic TeleSoft Ada Compiler Information

With optimization, the compilation process consists of 6 definite passes or steps:

Front-End:
Language control, translation to intermediate High-Form.
Middle Pass:
Resolution of tasking, translation to intermediate Low Form.
Optimizer:
Optimization of Low Form prior to code generation.
Code Generator:
Generation of native code for the target.
Prelinker (Binder):
Generation of elaboration code for main program.
Linker:
Linking the object modules generated by the compiler.

Compiler Option Information

B_TESTS:

```
ada -L -v -O D -a vme147.opt -m <main_unit> <test_name>
```

Legend:

ada	invoke Ada compiler
-L	generate interspersed source-error listing
-v	verbose
-O D	optimize with all options
- a vme147.opt	link options file
-m	designates main unit for driving the test
<test_name>	name of Ada source file to be compiled

Non_B Non-Family TESTS:

```
ada -v -O D -a vme147.opt -m <main_unit> <test_name>
```

Legend:

ada	invoke Ada compiler
-v	verbose
-O D	optimize with all options
- a vme147.opt	link options file
-m	designates main unit for driving the test
<test_name>	name of Ada source file to be compiled

Non_B Family TESTS:

```
ada -v -O D <test_name>
ald -a vme147.opt <main_unit>
```

Legend:

ada	invoke Ada compiler
-v	verbose
ald	invoke linker
-O D	optimize with all options
- a vme147.opt	link options file
<test_name>	name of Ada source file to be compiled

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	200 -- Value of V
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	"CCCCCCC10CCCCCCCC20CCCCCCCC30CCCCCCCC40 CCCCCCCC50CCCCCCCC60CCCCCCCC70CCCCCCCC80 CCCCCCCC90CCCCCCC100CCCCCCC110CCCCCCC120 CCCCCCC130CCCCCCC140CCCCCCC150CCCCCCC160 CCCCCCC170CCCCCCC180CCCCCCC190CCCCCC199"

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_646
$DEFAULT_MEM_SIZE	2147483647
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	TELEGEN2
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	ENT_ADDRESS
$ENTRY_ADDRESS1	ENT_ADDRESS1
$ENTRY_ADDRESS2	ENT_ADDRESS2
$FIELD_LAST	1000
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	4.25354E+37
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.80141E+38
$HIGH_PRIORITY	63

```
$ILLEGAL_EXTERNAL_FILE_NAME1
                        BADCHAR*^/%

$ILLEGAL_EXTERNAL_FILE_NAME2
                        /NONAME/DIRECTORY

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.ADA")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006E1.ADA")

$INTEGER_FIRST          -32768

$INTEGER_LAST           32767

$INTEGER_LAST_PLUS_1    32768

$INTERFACE_LANGUAGE     ASSEMBLY

$LESS_THAN_DURATION     -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -131_073.0

$LINE_TERMINATOR        ASCII.LF

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        M68K_CODE'(OP => NOP);

$MACHINE_CODE_TYPE      OPCODES

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2_147_483_648

$MIN_INT                -2147483648

$NAME                   NO_SUCH_TYPE_AVAILABLE

$NAME_LIST              TELEGEN2

$NAME_SPECIFICATION1    /plug2/ada/sh3.25/e68k/acvc/X2120A

$NAME_SPECIFICATION2    /plug2/ada/sh3.25/e68k/acvc/X2120B
```

```
$NAME_SPECIFICATION3   /plug2/ada/sh3.25/e68k/acvc/X3119A
$NEG_BASED_INT         16#FFFFFFFE#
$NEW_MEM_SIZE          2147483647
$NEW_STOR_UNIT         8
$NEW_SYS_NAME          TELEGEN2
$PAGE_TERMINATOR       ASCII.FF
$RECORD_DEFINITION     RECORD OP : OPCODES; END RECORD;
$RECORD_NAME           M68K_CODE
$TASK_SIZE             32
$TASK_STORAGE_SIZE     1024
$TICK                  2#1.0#E-14
$VARIABLE_ADDRESS      VAR_ADDRESS
$VARIABLE_ADDRESS1     VAR_ADDRESS1
$VARIABLE_ADDRESS2     VAR_ADDRESS2
```

APPENDIX B

COMPILATION SYSTEM AND LINKER OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

The options available with the *ada* command are summarized in Table 2-2. The default situation (that is, what happens if the option is not used) is explained in the middle column. Each option is described in the paragraphs that follow the table.

Table 2-2. Summary of Compiler Options

Option	Default	Discussed in Section
Common options:		
-l(ibfile <libname>	Use liblst.alb as the library file.	1.3.1
-t(emplib <sublib...>	None	1.3.1
-V(space_size <value>	Set size to 2000 Kbytes.	1.3.2
-v(erbose	Do not output progress messages.	1.3.3
-b(ind_only	Bind and link.	2.1.1
-c(pu_type <value>	Consider CPU type to be MC68020.	2.1.2
-d(ebug	Do not include debug information in object code. (*-d* sets -k(eep.)	2.1.3
-E(rror_abort <value>	Abort compilation after 999 errors.	2.1.4
-e(rrors_only	Run middle pass and code generator, not just front end.	2.1.5
-i(nhibit <key>†	Do not suppress run-time checks, source line references, or subprogram name information in object.	2.1.6
-k(eep	Discard intermediate representations of secondary units.	2.1.7
-m(ain <unit>	Do not produce executable code (binder/linker not executed).	2.1.8
-O(ptimize <key>†	Do not optimize code.	2.1.9
-s(oftware_float	Use hardware floating-point support.	2.1.10
-u(pdate_lib <key>†	Do not update library when errors are found (multi-unit compilations).	2.1.11
-x(ecution_profile	Do not generate execution-profile code.	2.1.12
Listing options:		
-C(ontext <value>	Include 1 line of context with error message.	2.1.13.1
-L(ist	Do not generate a source-error listing.	2.1.13.2
-F(ile_only_errs	Do not generate an errors-only listing.	2.1.13.3
-S(ource_asm	Do not generate assembly listing.	2.1.13.4

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are given on the following page.

Attachment F: Package Standard Information

For the SUN-4/E68 Ada compiler running on a SPARC based Sun-4 Server under Sun OS 4.01 Operating System, the numeric types and their properties are as follows:

Integer types:

INTEGER
size = 16
first = -32768
last = +32767

LONG_INTEGER
size = 32
first = -2147483648
last = +2147483647

Floating-point types:

FLOAT
size = 32
digits = 6
'first = -1.70141E+38
'last = +1.70141E+38
machine_radix = 2
machine_mantissa = 24
machine_emin = -125
machine_emax = +128

LONG_FLOAT
size = 64
digits = 15
'first = -1.79769E+308
'last = 1.79769E+308
machine_radix = 2
machine_mantissa = 53
machine_emin = -1021
machine_emax = +1024

Fixed-point types:

DURATION
size = 32
delta = 2#1.0#e-14
first = -86400
last = +86400

Table 3-4. Summary of LRM Chapter 13 Features for TeleGen2

13.1 Representation Clauses	Supported, except as indicated below (LRM 13.2 – 13.5). Pragma Pack is supported, *except for* dynamically sized components. For details on the TeleGen2 implementation of pragma Pack, see Section 3.7.1.
13.2 Length Clauses	Supported: 'Size 'Storage_Size for collections 'Storage_Size for task activation 'Small for fixed-point types See Section 3.7.2 for more information.
13.3 Enumeration Rep. Clauses	Supported, *except for* type Boolean or types derived from Boolean. (Note: users can easily define a non-Boolean enumeration type and assign a representation clause to it.)
13.4 Record Rep. Clauses	Supported *except for* records with dynamically sized components. See Section 3.7.4 for a full discussion of the TeleGen2 implementation.
13.5 Address Clauses	*Supported for:* objects (including task objects) and entries. *Not supported for:* packages, subprograms, or task units (Note: the TeleGen2 philosophy is to locate compilation units via the link process rather than in source programs.) See Section 3.7.5 for more information.
13.5.1 Interrupts	Supported. Some interrupt optimizations are supported as well. See "Using Machine Code Insertions" in the Programming Guide chapter for a full discussion of the TeleGen2 implementation of interrupts.
13.6 Change of Representation	Supported, *except for* types with record representation clauses.

------ *Continued on the next page* -----

Table 3-4. Summary of LRM Chapter 13 Features for TeleGen2 (Cont'd)

	—— Continued from the previous page ——	
13.7	Package System	Conforms closely to LRM model. Refer to Section 3.7.7 for details on the TeleGen2 implementation.
13.7.1	System-Dependent Named Numbers	Refer to the specification of package System (Section 3.7.7).
13.7.2	Representation Attributes	Implemented as described in LRM *except that:* 'Address for packages is unsupported. 'Address of a constant yields a null address.
13.7.3	Representation Attributes of Real Types	See Table 3-2.
13.8	Machine Code Insertions	Fully supported. The TeleGen2 implementation defines an attribute, 'Offset, that, along with the language-defined attribute 'Address, allows addresses of objects and offsets of data items to be specified in stack frames. Refer to "Using Machine Code Insertions" in the Programming Guide chapter for a full description on the implementation and use of machine code insertions.
13.9	Interface to Other Languages	Pragma Interface is supported for Assembly, C, UNIX, and Fortran. Refer to "Interfacing to Other Languages" in the Programming Guide chapter for a description of the implementation and use of pragma Interface.
13.10	Unchecked Programming	Supported except as noted below (LRM 13.10.2).
13.10.1	Unchecked Storage Deallocation	Supported.
13.10.2	Unchecked Type Conversions	Supported *except for* unconstrained record or array types.

Table 3-5. IBM Appendix F for TeleGen2

(1) Implementation-Dependent Pragmas	(a) Implementation-defined pragmas: Comment, Linkname, Integer, and No_Suppress (Section 3.8.1). (b) Predefined pragmas with implementation-dependent characteristics: • Interface (assembly, UNIX, C, and Fortran—see "Interfacing to Other Languages." Not supported for library units. • List and Page (in context of source/error compiler listings.) (See the *User Guide*.) • Pack. See Section 3.7.1. • Inline. Not supported for library-level subprograms. • Priority. Not supported for main programs. *Other supported predefined pragmas:* Controlled Shared Suppress Elaborate *Predefined pragmas partly supported* (see Section 3.1): Memory_Size Storage_Unit System_Name *Not supported:* Optimize
(2) Implementation-Dependent Attributes	'Offset. Used for machine code insertions. The predefined attribute 'Address is not supported for packages. See "Using Machine Code Insertions" earlier in this chapter for information on 'Offset and 'Address. 'Extended_Image 'Extended_Value 'Extended_Width 'Extended_Aft 'Extended_Digits Refer to Section 3.8.2 for information on the implementation-defined extended attributes listed above.
(3) Package System	See Section 3.7.7.
(4) Restrictions on Representation Clauses	Summarized in Table 3-4.

------- *Continued on the next page* -------

Table 3-5. LRM Appendix F for TeleGen2 (Contd)

——— Continued from the previous page ———	
(5) Implementation-Generated Names	None
(6) Address Clause Expression Interpretation	An expression that appears in an object address clause is interpreted as the address of the first storage unit of the object.
(7) Restrictions on Unchecked Conversions	Summarized in Table 3-4.

(8) Implementation-Dependent Characteristics of the I/O Packages.	1. In Text_IO, the type Count is defined as:
	`type Count is range 0 .. Text_IO_Definitions."-"(Text_IO_Definitions.Count'Last, 1);`
	`-- or 0..Max_Int-1`
	2. In Text_IO, the subtype Field is defined as:
	`subtype Field is Integer range 0 .. Text_IO_Definitions.Field_Last; -- or 0..1000`
	3. In Text_IO. the Form parameter of procedures Create and Open is not supported. (If you supply a Form parameter with either procedure, it is ignored.)
	4. Sequential_IO and Direct_IO cannot be instantiated for unconstrained array types or discriminated types without defaults.
	5. The standard library contains preinstantiated versions of Text_IO.Integer_IO for types Integer and Long_Integer and of Text_IO.Float_IO for types Float and Long_Float. We suggest that you use the following to eliminate multiple instantiations of these packages:
	`Integer_Text_IO`
	`Long_Integer_Text_IO`
	`Float_Text_IO`
	`Long_Float_Text_IO`

3.8.1. Implementation-Defined Pragmas. There are four implementation-defined pragmas in TeleGen2: pragmas Comment, Linkname, Images, and No_Suppress.

3.8.1.1. Pragma Comment. Pragma Comment is used for embedding a comment into the object code. Its syntax is:

```
pragma Comment ( <string_literal> );
```

where "<string_literal>" represents the characters to be embedded in the object code. Pragma Comment is allowed only within a declarative part or immediately within a package specification. Any number of comments may be entered into the object code by use of pragma Comment.

3.8.1.2. Pragma Linkname. Pragma Linkname is used to provide interface to any routine whose name can be specified by an Ada string literal. This allows access to routines whose identifiers not conform to Ada identifier rules.

Pragma Linkname takes two arguments. The first is a subprogram name that has been previously specified in a pragma Interface statement. The second is a string literal specifying the exact link name to be employed by the code generator in emitting calls to the associated subprogram. The syntax is:

```
pragma Interface ( assembly, <subprogram_name> );
pragma Linkname ( <subprogram_name>, <string_literal> );
```

If pragma Linkname does not immediately follow the pragma Interface for the associated program, a warning will be issued saying that the pragma has no effect.

A simple example of the use of pragma Linkname is:

```
procedure Dummy_Access( Dummy_Arg : System.Address );
pragma Interface (assembly, Dummy_Access );
pragma Linkname (Dummy_Access, "_access");
```

A note of caution: In the example above, the link name generated may be "__access" instead of "_access" if your native C compiler prepends an underscore to the specified link name.

3.8.1.3. Pragma Images. Pragma Images controls the creation and allocation of the image and index tables for a specified enumeration type. The image table is a literal string consisting of enumeration literals catenated together. The index table is an array of integers specifying the location of each literal within the image table. The length of the index table is therefore the sum of the lengths of the literals of the enumeration type: the length of the index table is one greater than the number of literals.

The syntax of this pragma is:

```
pragma Images(<enumeration_type>, Deferred);
    -- or --
pragma Images(<enumeration_type>, Immediate);
```

The default, Deferred, saves space in the literal pool by not creating image and index tables for an enumeration type unless the 'Image, 'Value, or 'Width attribute for the type is used. If one of these attributes is used, the tables are generated in the literal pool of the compilation unit in which the attribute appears. If the attributes are used in more than one compilation unit, more than one set of tables is generated, eliminating the benefits of deferring the table. In this case, using

```
pragma Images(<enumeration_type>, Immediate);
```

will cause a single image table to be generated in the literal pool of the unit declaring the enumeration type.

For a very large enumeration type, the length of the image table will exceed Integer'Last (the maximum length of a string). In this case, using either

```
pragma Images(<enumeration_type>, Immediate);
```

or the 'Image, 'Value, or 'Width attribute for the type will result in an error message from the compiler.

3.8.1.4. Pragma No_Suppress. No_Suppress is a TeleGen2-defined pragma that prevents the suppression of checks within a particular scope. It can be used to override pragma Suppress in an enclosing scope. No_Suppress is particularly useful when you have a section of code that relies upon predefined checks to execute correctly, but you need to suppress checks in the rest of the compilation unit for performance reasons.

Pragma No_Suppress has the same syntax as pragma Suppress and may occur in the same places in the source. The syntax is:

```
pragma No_Suppress (<identifier> [, [ON =>] <name>]);
```

where <identifier> is the type of check you *don't* want to suppress (e.g., access_check; refer to LRM 11.7)

<name> is the name of the object, type/subtype, task unit, generic unit, or subprogram within which the check is *not* to be suppressed; <name> is optional.

If neither Suppress nor No_Suppress are present in a program, no checks will be suppressed. You may override this default at the command level, by compiling the file with the -i(nhibit option and specifying with that option the type of checks you want to suppress. For more information on -i(nhibit, refer to your TeleGen2 *Overview and Command Summary* document.

If either Suppress or No_Suppress are present, the compiler uses the pragma that applies to the specific check in order to determine whether that check is to be made. If both Suppress and No_Suppress are present in the same scope, the pragma declared last takes precedence. The presence of pragma Suppress or No_Suppress in the source takes precedence over an -i(nhibit option provided during compilation.

3.8.2. Implementation-Dependent Attributes.

3.8.2.1. 'Address and 'Offset. These were discussed within the context of using machine code insertions, in the Programming Guide chapter.

3.8.2.2. Extended Attributes for Scalar Types. The extended attributes extend the concept behind the Text_IO attributes 'Image, 'Value, and 'Width to give the user more power and flexibility when displaying values of scalars. Extended attributes differ in two respects from their predefined counterparts:

1. Extended attributes take more parameters and allow control of the format of the output string.
2. Extended attributes are defined for all scalar types, including fixed and floating point types.

Extended versions of predefined attributes are provided for integer, enumeration, floating point, and fixed point types:

Integer:	'Extended_Image,	'Extended_Value,	'Extended_Width
Enumeration:	'Extended_Image,	'Extended_Value,	'Extended_Width
Floating Point:	'Extended_Image,	'Extended_Value,	'Extended_Digits
Fixed Point:	'Extended_Image, 'Extended_Aft	'Extended_Value,	'Extended_Fore,

The extended attributes can be used without the overhead of including Text_IO in the linked program. Below is an example that illustrates the difference between instantiating Text_IO.Float_IO to convert a float value to a string and using Float'Extended_Image:

```
with Text_IO;
function Convert_To_String ( Fl : Float ) return String is
Temp_Str : String ( 1 .. 6 + Float'Digits );
package Flt_IO is new Text_IO.Float_IO (Float);
begin
  Flt_IO.Put ( Temp_Str, Fl );
  return Temp_Str;
end Convert_To_String;

    function Convert_To_String_No_Text_IO( Fl : Float ) return String is
    begin
      return Float'Extended_Image ( Fl );
    end Convert_To_String_No_Text_IO;


with Text_IO, Convert_To_String, Convert_To_String_No_Text_IO;
procedure Show_Different_Conversions is
Value : Float := 10.03376;
begin
  Text_IO.Put_Line ( "Using the Convert_To_String, the value of the variable
  is : " & Convert_To_String ( Value ) );
  Text_IO.Put_Line ( "Using the Convert_To_String_No_Text_IO, the value
  is : " & Convert_To_String_No_Text_IO ( Value ) );
end Show_Different_Conversions;
```

3.8.2.2.1. Integer Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Width,Base,Based,Space_IF_Positive)

Returns the image associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that the value of Item is an integer literal with no underlines, no exponent, no leading zeros (but a single zero for the zero value), and a minus sign if negative. If the resulting sequence of characters to be output has fewer than Width characters, leading spaces are first output to make up the difference. (LRM 14.3.7:10,14.3.7:11)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more than one parameter. The parameter Item must be an integer value. The resulting string is without underlines, leading zeros, or trailing spaces.

Parameter Descriptions:

Item	The item for which you want the image: it is passed to the function. *Required*
Width	The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. *Optional*
Base	The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in base notation or not. If no preference is specified, the default (false) is assumed. *Optional*
Space_If_Positive	An indication of whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following subtype were declared:

```
subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Image(5)                    = "5"
X'Extended_Image(5,0)                  = "5"
X'Extended_Image(5,2)                  = " 5"
X'Extended_Image(5,0,2)                = "101"
X'Extended_Image(5,4,2)                = " 101"
X'Extended_Image(5,0,2,True)           = "2#101#"
X'Extended_Image(5,0,10,False)         = "5"
X'Extended_Image(5,0,10,False,True)    = " 5"
X'Extended_Image(-1,0,10,False,False)  = "-1"
X'Extended_Image(-1,0,10,False,True)   = "-1"
X'Extended_Image(-1,1,10,False,True)   = "-1"
```

```
X'Extended_Image(-1,0,2,True,True)      = "-2#1#"
X'Extended_Image(-1,10,2,True,True)     = "      -2#1#"
```

'Extended_Value

Usage:

X'Extended_Value(Item)

Returns the value associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that given a string, it reads an integer value from the beginning of the string. The value returned corresponds to the sequence input. (LRM 14.3.7:14)

For a prefix X that is a discrete type or subtype, this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Description:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type X. *Required*

Examples:

Suppose the following subtype were declared:

```
Subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Value("5")        = 5
X'Extended_Value(" 5")       = 5
X'Extended_Value("2#101#")   = 5
X'Extended_Value("-1")       = -1
X'Extended_Value(" -1")      = -1
```

'Extended_Width

Usage:

X'Extended_Width(Base,Based,Space_If_Positive)

Returns the width for subtype of X.

For a prefix X that is a discrete subtype. this attribute is a function that may have multiple parameters. This attribute yields the maximum image length over all values of the type or subtype X.

Parameter Descriptions:

Base	The base for which the width will be calculated. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether the subtype is stated in based notation. If no value for based is specified, the default (false) is assumed. *Optional*
Space_If_Positive	An indication of whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following subtype were declared:

```
Subtype X is Integer Range -10..16;
```

Then the following would be true:

```
X'Extended_Width                    = 3  -- "-10"
X'Extended_Width(10)                = 3  -- "-10"
X'Extended_Width(2)                 = 5  -- "10000"
X'Extended_Width(10,True)           = 7  -- "-10#10#"
X'Extended_Width(2,True)            = 8  -- "2#10000#"
X'Extended_Width(10,False,True)     = 3  -- " 16"
X'Extended_Width(10,True,False)     = 7  -- "-10#10#"
X'Extended_Width(10,True,True)      = 7  -- " 10#16#"
X'Extended_Width(2,True,True)       = 9  -- " 2#10000#"
X'Extended_Width(2,False,True)      = 6  -- " 10000"
```

3.8.2.2.2. Enumeration Type Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Width,Uppercase)

Returns the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that given an enumeration literal, it will output the value of the enumeration literal (either an identifier or a character literal). The character case parameter is ignored for character literals. (LRM 14.3.9:9)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more that one parameter. The parameter Item must be an enumeration value. The image of an enumeration value is the corresponding identifier, which may have character case and return string width specified.

Parameter Descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Width	The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. If the Width specified is larger than the image of Item, the return string is padded with trailing spaces. If the Width specified is smaller than the image of Item, the default is assumed and the image of the enumeration value is output completely. *Optional*
Uppercase	An indication of whether the returned string is in uppercase characters. In the case of an enumeration type where the enumeration literals are character literals, Uppercase is ignored and the case specified by the type definition is taken. If no preference is specified, the default (true) is assumed. *Optional*

Examples:

Suppose the following types were declared:

```
type X is (red, green, blue, purple);
type Y is ('a', 'B', 'c', 'D');
```

Then the following would be true:

```
X'Extended_Image(red)            = "RED"
X'Extended_Image(red, 4)         = "RED "
X'Extended_Image(red,2)          = "RED"
X'Extended_Image(red,0,false)    = "red"
X'Extended_Image(red,10,false)   = "red       "
Y'Extended_Image('a')            = "'a'"
Y'Extended_Image('B')            = "'B'"
Y'Extended_Image('a',6)          = "'a'  "
Y'Extended_Image('a',0,true)     = "'a'"
```

'Extended_Value

Usage:

X'Extended_Value(Item)

Returns the image associated with Item as defined in Text_IO.Enumeration_IO The Text_IO definition states that it reads an enumeration value from the beginning of the given string and returns the value of the enumeration literal that corresponds to the sequence input. (LRM 14.3.9:11)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Descriptions:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of X. *Required*

Examples:

Suppose the following type were declared:

```
type X is (red, green, blue, purple);
```

Then the following would be true:

```
X'Extended_Value("red")          = red
X'Extended_Value(" green")       = green
X'Extended_Value("    Purple")   = purple
X'Extended_Value(" GreEn  ")     = green
```

'Extended_Width

Usage:

X'Extended_Width

Returns the width for subtype of X.

For a prefix X that is a discrete type or subtype: this attribute is a function. This attribute yields the maximum image length over all values of the enumeration type or subtype X.

Parameter Descriptions:

There are no parameters to this function. This function returns the width of the largest (width) enumeration literal in the enumeration type specified by X.

Examples:

Suppose the following types were declared:

```
type X is (red, green, blue, purple);
type Z is (X1, X12, X123, X1234);
```

Then the following would be true:

```
X'Extended_Width     = 6  -- "purple"
Z'Extended_Width     = 5  -- "X1234"
```

3.8.2.2.3. Floating Point Attributes

'Extended_Image

Usage:

X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)

Returns the image associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameter Descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Fore	The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified, the default value (2) is assumed. *Optional*
Aft	The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. *Optional*
Exp	The minimum number of digits in the exponent. The exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. *Optional*
Base	The base that the image is to be displayed in. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true:

```
X'Extended_Image(5.0)                    = " 5.0000E+00"
X'Extended_Image(5.0,1)                  = "5.0000E+00"
X'Extended_Image(-5.0,1)                 = "-5.0000E+00"
X'Extended_Image(5.0,2,0)                = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)              = " 5.0"
X'Extended_Image(5.0,2,0,0,2)            = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)       = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)       = "2#1.1#E+02"
```

'Extended_Value

Usage:

X'Extended_Value(Item)

Returns the value associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it skips any leading zeros, then reads a plus or minus sign if present then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Descriptions:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of the input string. *Required*

Examples:

Suppose the following type were declared:

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true:

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
X'Extended_Value("2#1.01#E2")    = 5.0
```

'Extended_Digits

Usage:

X'Extended_Digits[(Base)]

Returns the number of digits using base in the mantissa of model numbers of the subtype X.

Parameter Descriptions:

Base	The base that the subtype is defined in. If no base is specified, the default (10) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is digits 5 range -10.0 .. 15.0;
```

Then the following would be true:

```
X'Extended_Digits     = 5
```

3.8.2.2.4. Fixed Point Attributes

'Extended_Image

Usage:

X'Extended_Image[(Item,Fore,Aft,Exp,Base,Based)]

Returns the image associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

For a prefix X that is a discrete type or subtype, this attribute is a function that may have more than one parameter. The parameter Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameter Descriptions:

Item	The item for which you want the image. It is passed to the function. *Required*
Fore	The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified, the default value (2) is assumed. *Optional*
Aft	The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. *Optional*
Exp	The minimum number of digits in the exponent; the exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default '3' is assumed. If Exp is 0, no exponent is used. *Optional*
Base	The base in which the image is to be displayed. If no base is specified, the default '10' is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default 'false' is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true:

```
X'Extended_Image(5.0)                 = " 5.00E-00"
X'Extended_Image(5.0,1)               = "5.00E-00"
X'Extended_Image(-5.0,1)              = "-5.00E-00"
X'Extended_Image(5.0,2,0)             = " 5.0E-00"
X'Extended_Image(5.0,2,0,0)           = " 5.0"
X'Extended_Image(5.0,2,0,0,2)         = "101.0"
X'Extended_Image(5.0,2,0,0,2,True,    = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)    = "2#1.1#E-02"
```

'Extended_Value

Usage:

X'Extended_Value(Image)

Returns the value associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it skips any leading zeros, reads a plus or minus sign if present, then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype, this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter Descriptions:

Image	Parameter of the predefined type string. The type of the returned value is the base type of the input string. *Required*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true:

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
X'Extended_Value("2_1.01_E2")    = 5.0
```

'Extended_Fore

Usage:

X'Extended_Fore(Base,Based)

Returns the minimum number of characters required for the integer part of the based representation of X.

Parameter Descriptions:

Base	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true:

```
X'Extended_Fore                 = 3  -- "-10"
X'Extended_Fore(2)              = 6  -- " 10001"
```

'Extended_Aft

Usage:

X'Extended_Aft(Base,Based)

Returns the minimum number of characters required for the fractional part of the based representation of X.

Parameter Descriptions:

Base	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared:

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true:

```
X'Extended_Aft      = 1  -- "1" from 0.1
X'Extended_Aft(2)   = 4  -- "0001" from 2#0.0001#
```

3.8.3. Package System. The current specification of package System is provided below.

```
package System is

  type Address is access integer;

  type Subprogram_Value is private;

  type Name     is (TeleGen2);

  System_Name   : constant name := TeleGen2;

  Storage_Unit : constant := 8;
  Memory_Size  : constant := (2 ** 31)  -1;

  -- System-Dependent Named Numbers:

  -- See Table 3-2 for the values for attributes of
  -- types Float and Long_Float

  Min_Int       : constant :=  -(2 ** 31);
  Max_Int       : constant := (2 ** 31)  -1;
  Max_Digits    : constant := 15;
  Max_Mantissa : constant := 31;
  Fine_Delta    : constant := 1.0 / (2 ** Max_Mantissa);
  Tick          : constant := 10.0E-3,

  -- Other System-Dependent Declarations

  subtype Priority is integer range 0 .. 63;

private
  -- ...
end System;
```